

CSC2/452 Computer Organization IO, OS, and Virtualization

Sreepathi Pai

URCS

October 28, 2019

Outline

Administrivia

Recap

System-level I/O

Time-sharing

The Operating System

Outline

Administrivia

Recap

System-level I/O

Time-sharing

The Operating System

Administrivia

- ▶ Assignment #3 (Memory Allocator) is out today
 - ▶ Due Nov 8, 2019 at 7PM
 - ▶ Start early, this one is difficult
 - ▶ 300 to 400 lines of pointer-heavy code
- ▶ Homework #5 out Wednesday
 - ▶ Due Monday, Nov 11

Outline

Administrivia

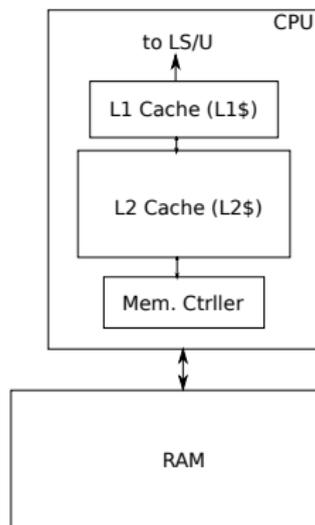
Recap

System-level I/O

Time-sharing

The Operating System

CPU + RAM



What about peripheral devices or input/output devices?

- ▶ Mouse, Keyboard
- ▶ Monitor
- ▶ Disks

Outline

Administrivia

Recap

System-level I/O

Time-sharing

The Operating System

Bus

A bus is a logical interface for communication between multiple devices. In its simplest form, it is a bunch of wires with devices connected to it (e.g. I2C).

- ▶ Peripheral Component Interconnect (PCI)
 - ▶ Succeeded ISA (Industry Standard Architecture, the bus in the IBM PC)
 - ▶ Most modern computers have PCIe (for express)
 - ▶ Used for “extension cards” (e.g. sound cards, video cards, etc.)
- ▶ Serial AT Attachment
 - ▶ AT: Advanced (1980s!) Technology (from IBM PC/AT)
 - ▶ Used for (internal) rotating hard disks
- ▶ Universal Serial Bus (USB)
 - ▶ Commonly used for external devices
- ▶ I2C
 - ▶ Used for sensors (fan speed, temperature, etc.)
 - ▶ More common in embedded devices (e.g. Raspberry Pi)

I/O Port

Since multiple devices can be connected to the same bus, we need a mechanism to address each device.

- ▶ Many different solutions, no one solution
- ▶ Common idea, assign a set of addresses to a device
 - ▶ Usually such an I/O address is also called an I/O port

x86 I/O

- ▶ On x86, I/O addresses look like memory addresses, but cannot be used by load/store instructions
- ▶ These I/O addresses form a distinct I/O address space
- ▶ x86 has special instructions to read/write I/O devices:
 - ▶ IN imm8 (or DX), AL or AX or EAX
 - ▶ OUT
 - ▶ also, for strings for data, INS and OUTS

Design considerations

Why a separate I/O address space?

Alternate Design: Memory-mapped I/O

- ▶ Devices appear in the same address space as RAM
- ▶ Use load/store instructions to read/write from these devices
- ▶ Processor redirects loads/stores to devices or to RAM
 - ▶ Depends on address of load/store
- ▶ Key advantage: no I/O instructions needed
- ▶ Key disadvantage: consumes part of address space
 - ▶ Not a big deal in 64-bit address spaces?

Setting up MMIO

- ▶ The OS and/or the firmware (usually BIOS) query each device connected to a bus
 - ▶ Usually at boot
- ▶ Each attached device is given a portion of the address space
 - ▶ This space cannot be used as “memory”
 - ▶ All reads/writes to these addresses will go to the device
- ▶ Needless to say:
 - ▶ This varies from system to system (PC/Mac/Raspberry Pi, etc.)
 - ▶ Also varies by operating system

Communicating with an I/O device

- ▶ In the early days of computing, I/O was very slow
 - ▶ Actually handed off to dedicated I/O processors
- ▶ I/O is still considered slow relative to the CPU
 - ▶ A CPU can execute billions of instructions per second
 - ▶ Keyboards, printers, scanners, disks, etc. are much slower
- ▶ Some I/O devices are very very fast
 - ▶ High-speed network devices (10GbE)
 - ▶ Some new flash memory
- ▶ Two strategies to deal with I/O devices
 - ▶ Polling (Synchronous)
 - ▶ Interrupt-Driven (Asynchronous)

Synchronous I/O (aka Polling)

- ▶ CPU sends request to device
 - ▶ By writing to an I/O port (or mapped address)
- ▶ CPU “polls” device for request completion
 - ▶ Usually by reading a status I/O port in a loop
 - ▶ Are you done yet?
 - ▶ Are you done yet?
 - ▶ Are you done yet?
 - ▶ ...
- ▶ When device completes request, the status value CPU is reading changes
- ▶ The CPU can then continue whatever it was doing

Polling

- ▶ Polling is simple, but ties up the CPU
- ▶ Usually avoided except for very high-speed devices
 - ▶ In these cases, devices can respond within nanoseconds
 - ▶ Examples: high-bandwidth network links (10GbE, 100GbE)

Asynchronous I/O (aka Interrupt-Driven I/O)

- ▶ CPU sends request to device
- ▶ Device begins working on request
- ▶ When device finishes, it *notifies* the CPU that it is done
 - ▶ Notification sent using a hardware mechanism called an *interrupt*
- ▶ CPU receives interrupt
 - ▶ Stops whatever it is doing
 - ▶ Handles the interrupt (i.e., the I/O request)
 - ▶ Resumes whatever it was doing when the interrupt arrived

Interrupt Service Routine (ISR)

- ▶ An ISR is the handler for the interrupt
- ▶ It is automatically invoked by the processor when an interrupt is received
 - ▶ On the x86, in 16-bit “real mode”, the first 1024 bytes of memory contains the function pointers for each ISR
 - ▶ Address 0 contains the address of ISR for Interrupt 0
 - ▶ Address 4 contains the address of ISR for Interrupt 1 and so on
 - ▶ 32 and 64-bit x86 allow this table of function pointers to be anywhere in memory
- ▶ An interrupt may occur at any time (asynchronous)
- ▶ How should an ISR save and resume the execution state?

Scenario: Knowing where to return

ISR:

```
... read value from I/O device into eax
test %eax, %eax
jz ...
...
iret
```

User's program:

```
add $1, %ecx
-----> interrupt happens here
cmp %ecx, $9
jg loop_exit
...
```

There is no CALL to the ISR, how does `iret` know where to return?

Scenario: Preserving State

ISR:

```
... read value from I/O device into eax
test %eax, %eax
jz ...
...
iret
```

User's program:

```
add $1, %ecx
cmp %ecx, $9
-----> interrupt happens here
jg loop_exit
...
```

What value of EFLAGS will `jg` operate on?

Execution State

It is possible to store the current “execution” state of the processor, by storing the current values of the following registers *before* switching to the ISR and restoring them after `iret`

- ▶ Program Counter (PC) (`rip`)
- ▶ Flags register (EFLAGS) (`eflags`)
- ▶ And in some special cases (read the x86 manual for IRET):
 - ▶ Segment Registers (`cs`, etc.)
 - ▶ Stack Pointer (SP) (`rsp`)
 - ▶ and other registers ...

The processor automatically saves the processor state before invoking the ISR.

Interrupts

- ▶ Interrupts originally were used by external devices to signal the CPU
- ▶ But the CPU also uses interrupts to signal exceptions:
 - ▶ division by zero,
 - ▶ segmentation fault
- ▶ Software can also generate interrupts directly
 - ▶ using the `int` instruction
 - ▶ for example, the Linux kernel system calls can be invoked using `int $0x80` in 32-bit mode
 - ▶ example of a system call: open a file

No Interruptions

- ▶ It is possible to turn off interrupts (or actually “mask” them)
 - ▶ Interrupts are still generated
 - ▶ But are not handled
- ▶ This is very useful when you do not want interference from ISRs
- ▶ However, some interrupts cannot be turned off (“non-maskable interrupts”)

Direct Memory Access (DMA)

- ▶ A DMA controller is a device that can read/write to RAM independent of the CPU
- ▶ Used to transfer large amounts of data from RAM to a hardware device (or vice versa)
 - ▶ Without occupying the CPU
- ▶ General flow:
 - ▶ CPU sets up DMA controller to read/write from one range of memory addresses to another
 - ▶ DMA performs the reads/writes
 - ▶ DMA interrupts the CPU when it is done

Summary of I/O

- ▶ Memory-mapped I/O
- ▶ Synchronous I/O (Polling)
- ▶ Asynchronous I/O (Interrupt-driven I/O)
 - ▶ Interrupts, ISRs, etc.
- ▶ DMA

Outline

Administrivia

Recap

System-level I/O

Time-sharing

The Operating System

A Hardware Timer

- ▶ Imagine you have a timer device that generates interrupts at a fixed rate
 - ▶ Say one every 100ms
- ▶ Can be used to keep track of time at 100ms intervals
- ▶ Or can be used to implement a crude form of time-sharing
 - ▶ Where multiple programs run at the same time

Crude Time-sharing

- ▶ Program A is running
- ▶ Timer interrupt fires, and control transfers to ISR
- ▶ ISR saves Program A's state, and returns to Program B
 - ▶ Does not use IRET
- ▶ Repeat by returning to Program C, Program D, Program A, Program B, etc.

Non-crude Time-sharing

- ▶ Most modern operating systems use the hardware clock to give each program a slice of CPU time
 - ▶ Usually 100ms
- ▶ Program state is saved and another program is given a time slice at the end of each interval
 - ▶ CPU supports special instructions to save/restore program state
- ▶ This is the first essential step to get multiple programs running on the CPU
 - ▶ Called “CPU virtualization”
 - ▶ Each program thinks it has use of full CPU

Outline

Administrivia

Recap

System-level I/O

Time-sharing

The Operating System

Resources

- ▶ Many resources on a computer
 - ▶ CPU
 - ▶ RAM
 - ▶ I/O devices
- ▶ How to share these resources among different programs?
 - ▶ For efficiency
- ▶ How to provide uniform interfaces for different devices?
 - ▶ For productivity

Virtualization

- ▶ The Operating System (OS) and the CPU conspire to make every program believe it has sole access to the machine.
- ▶ Each program believes
 - ▶ it has its own CPU
 - ▶ RAM
 - ▶ access to I/O devices
- ▶ Except that behind the scenes, multiple programs are:
 - ▶ time-sharing the CPU
 - ▶ space-sharing/time-sharing the RAM
 - ▶ interleaving accesses to I/O devices

The Operating System

- ▶ Also called the kernel
- ▶ A program that interfaces to the CPU and I/O devices on behalf of user programs
- ▶ Provides uniform abstractions over diverse hardware
 - ▶ A keyboard is a keyboard, regardless of whether it is bluetooth, USB or PS/2
- ▶ Provides a large number of useful services to programs
 - ▶ Data/file storage, network connectivity, etc.
- ▶ Security mechanisms
- ▶ Learn more about OS in CSC256
 - ▶ In 252, we will focus on OS + CPU/Hardware interactions

Process

A process is an operating system abstraction for a (running) program and its state.

- ▶ Code
- ▶ Data
- ▶ All CPU state (program counter, stack, etc.)
- ▶ All files
- ▶ Network connections, etc.

Different processes share the CPU using time-sharing (also called multitasking).

Signals

A signal is a Unix-specific mechanism to notify programs asynchronously.

- ▶ It is a software analogue to interrupts
 - ▶ Actually, many interrupts are translated into signals
- ▶ A handler is a C function that is called when a signal is received
 - ▶ Equivalent to an ISR
- ▶ The OS:
 - ▶ Saves the current execution state of the program
 - ▶ Calls the signal handler
 - ▶ Resumes execution once the signal handler returns

Generating Signals

- ▶ The `raise` or `kill` system calls can be used to generate signals
 - ▶ Analogous to the `Int` instruction
- ▶ A program can send signals to itself using `raise`
- ▶ A program can send signals to other programs using `kill`
 - ▶ A crude form of interprocess communication
 - ▶ And clearly not a very well-thought out name

Some examples of signals

- ▶ C standard signals (`signal.h`)
 - ▶ SIGABRT, abort execution
 - ▶ SIGFPE, floating point exception
 - ▶ SIGILL, invalid/illegal instruction
 - ▶ SIGINT, interrupt program, sent when you press CTRL+C
 - ▶ SIGSEGV, segmentation fault
 - ▶ SIGTERM, terminate request, sent when you press CTRL+\
- ▶ Other important Unix signals
 - ▶ SIGSTOP, stop a program
 - ▶ SIGCONT, continue a stopped program
 - ▶ SIGTSTP, request a program to stop, sent when you press CTRL+Z
 - ▶ SIGKILL, kill a program
 - ▶ many more ..., run `kill -l` in a shell for the list

Signal handling

- ▶ Most signals can be “caught” and handled, or ignored
 - ▶ You provide handlers for them, or use default handler
- ▶ Some signals cannot be caught or ignored
 - ▶ SIGKILL
 - ▶ SIGSTOP

Example: Handler

```
void my_int_handler(int sig, siginfo_t *info, void *ucontext) {  
    puts("In interrupt handler, receiving signal\n");  
}
```

- ▶ `sig` is the signal being handled
- ▶ `info` contains lots more information about where and how the signal was generated
- ▶ `ucontext` contains information on what the program was doing
 - ▶ Not generally used

Example: Setting up a handler

```
int main(void) {
    struct sigaction setup;
    struct sigaction prev;

    // Indicate that I'm providing a new handler
    setup.sa_flags = SA_SIGINFO;

    // Provide the function pointer to my new handler
    setup.sa_sigaction = my_int_handler;

    // install handler for sigint
    if(sigaction(SIGINT, &setup, &prev) == 0) {
        int x;

        while(1) {
            printf("Enter a number: ");
            scanf("%d", &x);
        }
    } else {
        printf("ERROR: Could not install signal handler: %d\n", errno);
    }
}
```

Demo

Slide intentionally blank.

scanf

Why did `scanf` terminate?

In the `scanf` manual page:

`EINTR` The read operation was interrupted by a signal; see `signal(7)`.

References

- ▶ For hardware details
 - ▶ Read the Intel manuals
- ▶ For Processes
 - ▶ Section 8.2 of the textbook
- ▶ For Signals
 - ▶ Section 8.5 of the textbook
- ▶ Next class: Virtual Memory (Chapter 8)